Ejecución sincrónica vs. ejecución asincrónica

Introducción

Como ya vimos en el primer apunte de manejo de archivos, NodeJS nos ofrece múltiples funciones para realizar operaciones sobre archivos con mucha facilidad. Ya hemos visto las versiones **sincrónicas** y nos ha quedado pendiente ver las **asincrónicas**. Pero ¿Qué significa que una función sea *asincrónica*? Veamos un poco de qué se trata.

Ejecución sincrónica

Siempre que escribamos más de una instrucción en un programa, acostumbramos esperar que las instrucciones se ejecuten comenzando desde la primera línea, una por una, de arriba hacia abajo, hasta llegar al final del bloque de código. En el caso de que una de esas instrucciones sea una llamada a otra función, el orden de ejecución se pausa, y se procede a ejecutar las instrucciones dentro de esa función. Sólo una vez ejecutadas todas las instrucciones de esa función, es que el programa retomará con el flujo de instrucciones que venía ejecutando antes.

En todo momento, **sólo se están ejecutando las instrucciones de una sola de las funciones** a la vez. O sea, debe finalizar una función para poder continuar con la otra.

Siguiendo esta idea, el fin de una función marca el inicio de la siguiente, y el fin de ésta, el inicio de la que le sigue, y así sucesivamente, describiendo una secuencia que ocurre en *una única línea de tiempo*.

A está forma de ejecución se la conoce como ***sincrónica***.

Ejemplo

```
function funA(){
    console.log(1)
    funB()
    console.log(2)
}

function funB(){
    console.log(3)
    funC()
    console.log(4)
}
```

```
function funC(){
    console.log(5)
}
```

Al ejecutar la función `funA()` se muestra lo siguiente por pantalla:

```
1
3
5
4
2
```

Comportamiento de una función: bloqueante vs no-bloqueante

Cuando alguna de las instrucciones dentro de una función intente acceder a un recurso que se encuentre fuera del programa (por ejemplo, enviar un mensaje por la red, o leer un archivo del disco) nos encontraremos con dos maneras distintas de hacerlo: en forma bloqueante, o en forma no-bloqueante (*blocking* o *non-blocking*).

Operaciones bloqueantes

Este tipo de operaciones permiten que el programa se comporte de la manera más intuitiva, es decir, siguiendo las reglas establecidas en el punto anterior (ejecución sincrónica).

De esta manera si quisiéramos, por ejemplo, grabar un texto dentro de varios archivos, podríamos hacer lo siguiente:

```
const texto = 'un texto largo [...] muy largo'
const archivos = ['f1.txt', 'f2.txt', 'f3.txt', 'f4.txt']

for (const arch of archivos) {
    fs.writeFileSync(arch, texto)
}
```

Imaginemos que es un texto realmente largo, y que la operación de grabado tarda unos... 2 segundos por archivo. Podemos simular este comportamiento importando el siguiente módulo:

```
const sleep = require('sleep')
```

y utilizando la función `sleep.sleep(segundos)`. Esta función recibe como parámetro la cantidad de segundos que debe esperar antes de permitir que el programa se continúe ejecutando.

Creando objetos de la clase `Date` (clase nativa de NodeJS), podemos obtener la fecha completa en un determinado instante. Esto nos servirá para monitorear el tiempo de inicio y fin, y poder medir la duración total de nuestro proceso.

Agregando todo esto, más algunos mensajes, nuestro pequeño programa se verá entonces de la siguiente manera:

```
const fs = require('fs')
const sleep = require('sleep');

const texto = "un texto largo laaaaaargo [...] muy largo!"
const archivos = ['f1.txt', 'f2.txt', 'f3.txt', 'f4.txt']

const timeStart = new Date()
console.log('comenzando...')

for (const arch of archivos){
    fs.writeFileSync(arch, texto)
    sleep.sleep(2)
    console.log(arch + ' grabado con exito')
}
const timeEnd = new Date()
const duracion = timeEnd - timeStart

console.log('finalizado en: %d segundos', duracion / 1000)
```

Si lo ejecutamos, obtendremos un resultado similar a éste:

```
comenzando...
f1.txt grabado con éxito
f2.txt grabado con éxito
f3.txt grabado con éxito
f4.txt grabado con éxito
finalizado en: 8.016 segundos
```

Operaciones no-bloqueantes

Ahora bien, en algunos casos esperar a que una operación termine para iniciar la siguiente puede no ser la mejor idea, ya que esto podría causar grandes demoras en la ejecución del programa. Es por eso que NodeJS ofrece una segunda opción: las operaciones no bloqueantes.
Este tipo de operaciones permite que, una vez iniciadas, el programa pueda continuar con la siguiente instrucción, sin esperar a que finalice la anterior. O sea, permite la ejecución de varias operaciones **en paralelo**, sucediendo al mismo tiempo. A este tipo de ejecución se la conoce como **asincrónica**.

Ejecución asincrónica

Para poder usar funciones que realicen operaciones no bloqueantes debemos aprender a usarlas adecuadamente, y no generar efectos adversos en forma accidental.

Cuando se trata de código que se ejecuta en forma sincrónica, establecer el orden de ejecución se vuelve tan fácil como decidir qué instrucción escribir primero. Sin embargo, cuando se trata de ejecución asincrónica, sólo sabemos en qué orden comenzarán su ejecución las instrucciones, pero no sabemos en qué momento ni en qué orden terminarán de ejecutarse.
Para ver un poco mejor el funcionamiento de este tipo de funciones, usaremos la siguiente función de NodeJS:

```
setTimeout(funcion, milis)
```

La función ***setTimeout*** recibe como primer parámetro otra función, que deseamos que se ejecute, y como segundo parámetro, una cantidad de tiempo (en milisegundos) que deseamos que se espere antes de resolver la función enviada como primer parámetro.

Ejemplo de uso:

```
function buenDia(){
    console.log('buen dia')
}

function buenasTardes(){
    console.log('buenas tardes')
}

function buenasNoches(){
    console.log('buenas noches')
}
```

```
setTimeout(buenDia, 1000)
setTimeout(buenasTardes, 2000)
setTimeout(buenasNoches, 3000)
```

Salida por pantalla:

```
buen dia
buenas tardes
buenas noches
```

Cada uno de estos mensajes aparecerá un segundo dsp del anterior.
Sin embargo, si ejecutamos las instrucciones de esta manera...

```
setTimeout(buenasNoches, 3000)
setTimeout(buenasTardes, 2000)
setTimeout(buenDia, 1000)
```

...igualmente obtendremos el mismo resultado! Esto es porque la función `setTimeout` realiza una operación no-bloqueante, por lo que las tres funciones se ejecutan casi al mismo tiempo (en realidad, una dsp de la otra, con diferencia de microsegundos, dependiendo de la velocidad del procesador). Sin embargo, cada una espera una determinada cantidad de tiempo antes de resolverse. Por ese motivo es que luego de 1 segundo, veremos el texto ***`buen dia`*** en pantalla, aunque está sea la última instrucción escrita en nuestro programa.

Pueden imaginarse qué se mostraría por pantalla si agregaramos la siguiente instrucción al final del programa?

```
setTimeout(buenasNoches, 3000)
setTimeout(buenasTardes, 2000)
setTimeout(buenDia, 1000)
console.log('fin')
```